TP02 - Estrutura de dados

Marcos Daniel Souza Netto - 2022069492

December 3, 2023

1 Introdução

O problema proposto consiste em implementar um programa que realiza q operações de dois tipos sobre um vetor de Matrizes 2x2: O primeiro tipo consiste em uma atualização de uma posição do vetor, e o segundo tipo consiste em uma consulta sobre um intervalo do vetor (Deve ser retornado a multiplicações de todas as matrizes do intervalo e depois aplicar essa transformação linear a um vetor dado pelo usuário).

Note que uma solução ingênua para o problema seria: Dado a operação de atualização i e a matriz M, basta atualizar a posição i do vetor com a matriz M (Custo constante). E para cada operação de consulta, basta percorrer o intervalo do vetor, multiplicando as matrizes e retornando o resultado e então multiplicá-lo pelo vetor. Porém, essa solução não é eficiente, pois uma operação de consulta desse tipo possui complexidade $O(n)$, e tendo em vista que vão ser realizadas um número grande de consultas, a complexidade passaria para $O(n \cdot q)$, onde q é o número de consultas.

A solução implementada e também proposta pelos professores consiste em utilizar a ideia de *Segment Tree*, que é uma estrutura de dados que permite realizar consultas em intervalos do vetor de forma eficiente.

2 Método

A seguir serão detalhados a implementação da solução, de modo a explicitar as estruturas de dados utilizadas e as estratégias de solução. A priori vale apresentar as especificações do ambiente de desenvolvimento utilizado:

- Sistema Operacional: Ubuntu 22.04.3 LTS;
- Compilador: G++ 11.4.0;
- Processador: Ryzen 5 5500u;
- Memória RAM: 8GB.

2.1 Estruturas de dados

Neste trabalho, foi utilizado apenas a estrutura de dados de *Segment Tree*, que foi implementada no arquivo *segtree.hpp*.

Como já mencionado anteriormente, essa estrutura de dados otimiza a consulta de intervalos de um vetor, e portanto, foi utilizada para otimizar a consulta de intervalos do vetor de matrizes.

A ideia é que cada nó da árvore representa um intervalo do vetor, e cada nó guarda o resultado da operação associada ao intervalo que ele representa e podemos combinar esses pedaços pré-computados para formar o resultado de uma consulta específica. Neste problema, cada nó guarda o resultado da multiplicação das matrizes do intervalo que ele representa. Assim, para realizar uma consulta, basta percorrer a árvore, e para cada nó, verificar se o intervalo que ele representa está contido no intervalo da consulta, caso esteja, basta retornar o valor guardado no nó, caso contrário, é necessário percorrer os filhos do nó, e repetir o processo. Com esses pedaços, podemos combiná-los (aqui, multiplicá-los) para encontrar o resultado. Note que nessa estrutura de dados é importante que a operação a ser realizada deve ter a propriedade de associatividade (A ordem em que as operações são realizadas não altera o resultado), e no caso da multiplicação de matrizes, essa propriedade é satisfeita.

Para não entrar nos detalhes dessa estrutura de dados, a implementação utilizada foi baseada na aula 9 da Maratona UFMG, disponível em: `https://youtu.be/OW_nQN-UQhA?si=4qCz2jhB6BHWVRMz`.

Propriedades da *Segment Tree*

- A altura da árvore é $O(\log n)$, onde n é o número de elementos do vetor;
- O número total de nós da árvore é $2n - 1$, onde n é o número de elementos do vetor;
- Para a consulta de um intervalo na árvore, visitamos no máximo 4 nós por nível, temos então $O(4 \cdot \log n) = O(\log n)$, onde n é o número de elementos do vetor;

Complexidade

- Para a construção da estrutura temos complexidade $O(n)$;
- Para a consulta na estrutura temos complexidade $O(\log n)$;
- Para a atualização de uma posição do vetor temos complexidade $O(\log n)$;

2.2 Classes e principais funções

O código foi dividido nos arquivos *matrix2.hpp*, *segtree.hpp* e o arquivo principal *main.cpp*.

- **matrix2.hpp**: Encapsulamento do tipo abstrato de dados de matriz 2x2, com as operações de multiplicação e atribuição;

  Matrix2(): O construtor padrão desse TAD inicializa a matriz com a matriz identidade;

  Matrix2(int, int, int, int): O construtor desse TAD recebe os valores dos elementos da matriz;

  operator*: Sobrecarga do operador de multiplicação, que recebe uma matriz e retorna a multiplicação das duas matrizes;

- **segtree.hpp**: Implementação da estrutura de dados árvore de segmentos;

  query: Função que recebe o intervalo da consulta e retorna a multiplicação das matrizes do intervalo;

  update: Função que recebe a posição do vetor a ser atualizada e a nova matriz, e atualiza a posição do vetor com a nova matriz, recalculando os valores dos nós da árvore que foram afetados pela atualização;

- **main.cpp**: Arquivo principal do programa que recebe os dados de entrada, os valida, e faz chamada para as funções de atualização e consulta da árvore de segmentos;

  main: Função principal do programa, que recebe os dados de entrada, os valida, faz chamada para as funções de atualização e consulta da árvore de segmentos e mostra o resultado das consultas na saída padrão;

3 Análise de Complexidade

A seguir serão apresentadas as análises de complexidade das principais funções do programa utilizadas na solução. Como a parte mais importante é a implementação da árvore de segmentos, a análise de complexidade será feita para as funções de construção, consulta e atualização da árvore de segmentos.

3.1 Construtor

A forma implementada para a construção da árvore de segmentos foi a inicialização de todas as posições do vetor que a representa com a matriz identidade. Dessa forma, como o tamanho do vetor é $4n+1$, onde n é o número de transformações lineares informada na entrada, temos que a complexidade da construção da árvore é $O(n)$.

3.2 Função *query*

Como mencionada na seção 2.1, a complexidade da consulta na árvore de segmentos é $O(\log n)$, onde n é o número de elementos do vetor. Isso acontece porque são necessário acessar no máximo 4 nós por nível da árvore para a construção da consulta, e a altura da árvore é $O(\log n)$.

3.3 Função *update*

De forma semelhante à função *query*, para atualizar a árvore com os intervalos já pré-computados, temos que atualizar uma folha da árvore de segmentos e em seguida subir na estrutura recalculando os valores dos nós pais. Poranto, temos a complexidade de atualização como $O(\log n)$.

3.4 Complexidade geral

Analisando o programa como um todo, temos que :

1. A complexidade da construção da árvore de segmentos é $O(n)$ que é realizada uma única vez;
2. A complexidade de uma operação, tanto de consulta quanto de atualização, é $O(\log n)$, e é realizada q vezes;

Portanto, a complexidade de tempo do programa é $O(n+q \cdot \log n)$, onde n é o número de transformações lineares informada na entrada e q é o número de consultas informada na entrada. Já a complexidade de espaço é $O(n)$, pois é necessário armazenar a árvore de segmentos, que neste caso tem o tamanho de $4n + 1$.

4 Estratégias de robustez

Com a utilização da biblioteca *msgassert*, presente no arquivo *msgassert.h*, foi implementada a verificação do domínio dos dados de entrada, especificados na descrição do trabalho. Quando o dado de entrada não respeita o domínio especificado, o programa é encerrado e uma mensagem do erro específico é exibida na saída padrão de erro.

5 Análise Experimental

A seguir serão apresentados os resultados obtidos na experimentação do programa.

5.1 Localidade de Referência

Utilizando o programa *analisamem*, fornecido pelos professores, foi possível observar a localidade de referência do programa. Para isso, foi utilizado o programa de geração de casos de testes, também fornecido pelos professores, para gerar uma entrada que realiza 1000 operações (consulta e atualização) em um vetor de 1000 posições.

Além disso, é importante salientar que toda a análise de localidade será feita sobre a estrutura principal do trabalho, a árvore de segmentos.

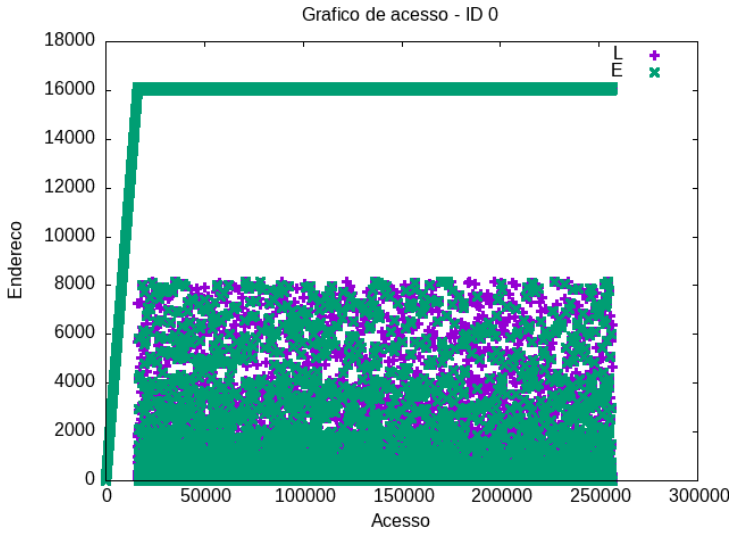

Figure 1: Leitura e escrita na memória

Essa figura é bastante interessante e informativa, pois mostra por meios práticos o funcionamento de uma árvore de segmentos: Note que a base do "retângulo" da figura é mais preenchida do que sua parte superior, isso se deve porque os nós mais próximos da raiz, na implementação utilizada, estão em endereços mais baixos enquanto as folhas estão em endereços mais altos.

- Em consultas, o intuito de se utilizar árvore de segmentos é exatamente utilizar de cálculos já precomputados e assim acessar menos os valores orginais do próprio vetor. Desse modo, utiliza-se dos valores que estão em endereços mais baixos no gráfico. Eventualmente, caso necessário, a estrutura deve descer para posições próximas das folhas, mas como mostrado no gráfico, ocorre com menos frequência.
- Em atualizações, o comportamento é semelhante ao anterior, com a diferença que obrigatoriamente é necessário descer para as folhas e assim recalculando os nós que estão no caminho da folha atualizada até a raiz. É fácil ver que também temos acessos mais frequentes para nós próximos da raiz afinal esse é o comportamento da própria operação de atualização.

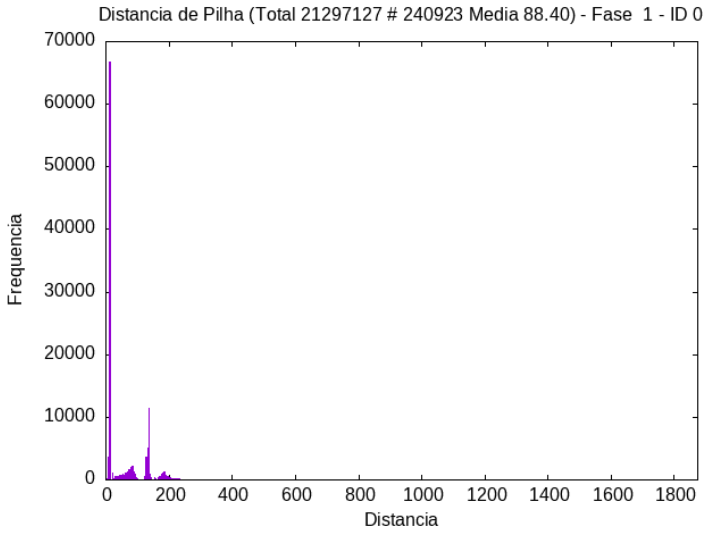

Figure 2: Distâcia de Pilha

A figura acima reafirma o argumento anterior: Alguns nós são acessados muito mais vezes do que outros. Isto é, nós mais proximos da raiz, devido o próprio comportamento da árvore de segmentos, são acessados com maior frequência.

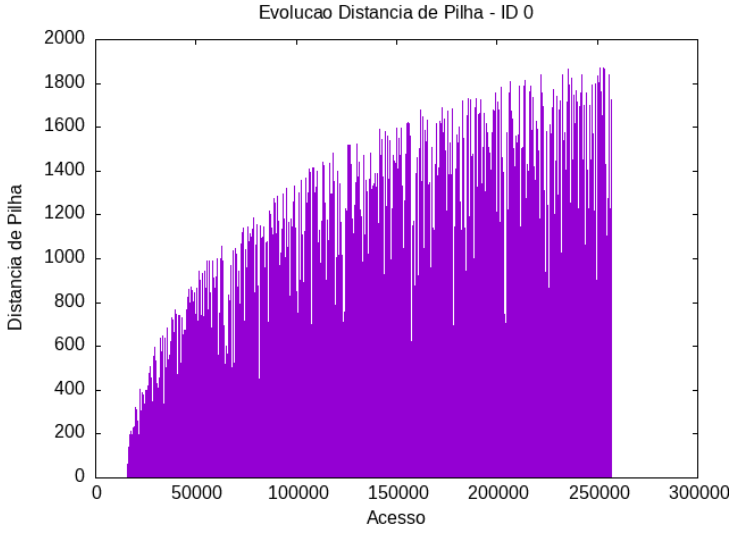

Figure 3: Evolução de Distância de Pilha

6 Conclusões

O trabalho proposto foi de grande valia para o aprendizado de estruturas de dados, e de como elas podem ser utilizadas para resolver de forma muito mais eficientes certos problemas. Neste caso, a estrutura de árvore de segmentos provê de forma eficiente operações de atualização e consulta sobre um vetor cuja operações sobre o dado tem propriedade associativa.

Bibliografia

- Slides da disciplina de Estrutura de Dados, ministrada pelo Prof. Wagner Meira Jr. e Prof. Eder Fereira Figuiredo.
- Aula 9 - SegTree, Maratona UFMG.

  Disponível em: `https://youtu.be/OW_nQN-UQhA?si=4qCz2jhB6BHWVRMz`

Instruções para compilação e execução

É necessário ter instalado na máquina o compilador g++, com pelo mínimo a versão descrita no começo do trabalho.

Em um terminal, navegue até a pasta raiz do projeto e execute os seguintes comandos:

```
$ make
$ ./tp03.out
```

Agora, basta seguir o padrão de entrada descrito na especificação do trabalho.